Mistore: A distributed storage system leveraging the DSL infrastructure of an ISP

Pierre Meye, Philippe Raipin, Frédéric Tronel, Emmanuelle Anceaume

Mistore: A distributed storage system leveraging the DSL infrastructure of an ISP

Pierre Meye*, Philippe Raipin*, Frédéric Tronel†, and Emmanuelle Anceaume‡
*Orange Labs, France, {pierre.meye, philippe.raipin}@orange.com
†Supelec, France, frederic.tronel@supelec.fr
‡CNRS / IRISA, France, anceaume@irisa.fr

***Abstract*—Internet Service Providers furnishing cloud storage services usually rely on big data centers. These centralized architectures induce many drawbacks in terms of scalability, reliability, and high access latency as data centers are single points of failure and are not necessarily located close to the users. This paper introduces Mistore, a distributed storage system aiming at guaranteeing data availability, durability, low access latency by leveraging the Digital Subscriber Line infrastructure of an ISP. Mistore uses the available storage resources of a large number of home gateways and points of presence for content storage and caching facilities reducing the role of the data center to a load balancer. Mistore also targets data consistency by providing multiple types of consistency criteria on content and a versioning system allowing users to get access to any prior versions of their contents. Mistore validation has been achieved through extensive simulations**

I. Introduction

Most of the existing architectures for data storage of Internet Service Providers (ISPs) are based on very large centralized datacenters that store and manage all the information related to their clients and their data. With the ever growing number of clients and the amount of data, centralized architectures reach their limit in terms of scalability, reliability, and access latencies. These drawbacks are commonly addressed by decentralizing the system over multiple geo-distributed nodes. Data are then placed on nodes close to users which reduces data access latencies and improves the scalability and reliability of the data by eliminating the single point of failure issue. Most of the storage systems addressing these issues either rely on peer to peer (P2P) technologies where peers are responsible for storing a subset of the data, or take advantage of the presence of large datacenters. In the former case, transient peers availability requires expensive maintenance procedures to reach an acceptable overall availability, while in the latter case, clients suffer from high data access latencies due to the remoteness of datacenters from users. This has recently led to the design of hybrid approaches relying on both peers and datacenters, where peers reduce datacenters workload and data access latencies, and improve system scalability while datacenters compensate peers instability.

In this paper, we investigate the design of a hybrid distributed storage system by leveraging highly available nodes in the network infrastructure of an ISP. Specifically, our approach combines the use of datacenters with home gateways and points of presence (POPs) equipments. Let us first briefly describe the Digital Subscriber Line (DSL) of an ISP. Fig. 1 illustrates a simple but common network infrastructure providing access to Internet through a DSL technology. Each user is equipped with a home gateway that provides access to multiple services like telephone, television and Internet. Each line of subscribers is first aggregated by a Digital Subscriber Line Access Multiplexer (DSLAM) which can aggregate thousands of lines. At a second level, the subscriber lines are aggregated in a high-capacity Asynchronous Transfer Mode (ATM) or Gigabit Ethernet link from the DSLAM to the IP network of the ISP. The flow coming from a DSLAM enters the ISP network via a POP that can handle links from a large number of DSLAMs. The ISP network is directly connected to the Internet backbone, and uses large datacenters to store information related to their subscribers and their data when a storage service is provided.

Based on this classical infrastructure, we aim at leveraging the Home Gateways (HGs) to take advantage of their native resources (*i.e.* computing, memory, and storage)[a], and to the fact that users let most of the time their HGs powered on [1]. Thus, exploiting resources provided by HGs should yield a large number of high available and intelligent storage nodes located very close to the users. Second, we also aim at leveraging the POPs to benefit from their natural geographic repartition and their position at the edge of the ISP network, and the fact that all the traffic within and across ISPs goes through the POPs. We plan to use the POPs to bring the intelligence of the storage system close to users, as pioneered in [2] for content caching and distribution in Content Delivery Networks (CDNs).

This paper presents Mistore, a distributed storage system dedicated to users who access Internet via a DSL technology that ensures data availability, durability, and low access latency by fully exploiting HGs storage capacities, POPs physical localizations, and datacenters. The focus of the paper is on how data consistency and replication are managed in Mistore.

[a]For instance the livebox play (http://liveboxplay.orange.fr/) of the french ISP Orange contains a 1.2GHz Intel Atom CE4257 processor, 2GB of RAM, and a hard drive of 320GB.

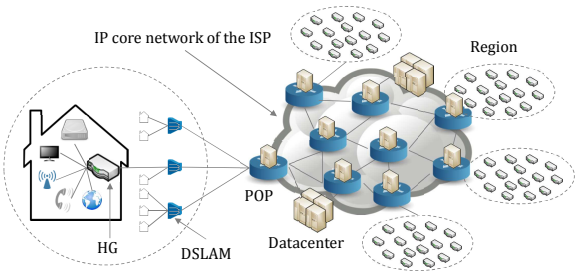

Figure 1. Simple overview of a DSL infrastructure of an ISP.

Data consistency: System designers usually face the CAP theorem (tradeoffs about Consistency, Availability and network Partition tolerance) when designing a distributed storage system [3], [4]. We choose to serialize the write requests in order to avoid the complexity of conflicts resolution to application developers. On the other hand multiple consistency criteria to parametrize the read requests are provided, namely a readers/writer mutual exclusion, an atomic consistency, and an eventual monotonic-read consistency criteria. We also provide a versioning system on data. This mechanism creates a new version of a piece of data when an update (*i.e.* a write request) is performed on it. This mechanism naturally provides a linear history about data and their updates.

Data replication: We ensure data availability and durability through replication. Users send their write requests to the closest POP that will perform the data replication on behalf to them. It aims to mitigate the bottleneck that users low upload bandwidths may cause by moving the replication overhead to the IP core network of the ISP where the bandwidth is much larger. To reduce latencies data are replicated synchronously on POPs and asynchronously on HGs and stripping techniques can be used to store and retrieve the data fragments.

The remainder of the paper is organized as follows. Section II presents related works. Section III presents the system model and the assumptions considered. The architecture is presented in section IV. The different consistency criteria are defined in Section V and the way we implement it as well as read and write operations is described in Section VI. Section VII presents the evaluation of our design choices and we conclude in Section VIII.

II. Related Work

Architecture: Recently, some works have investigated new designs of distributed storage systems to face dynamicity issues of pure P2P systems and the cost on large data centers [5]. The hybrid approaches, sometimes called peer-assisted approaches [6] are promising and are developed even for content distribution purpose [7]. Some works investigate the use of distributed and stable small datacenters or home gateways [1]. In contrast to those approaches, Mistore is dedicated to an ISP which is a clear advantage over other providers because they master all the network infrastructure and thus they can push contents closer to the users [8].

Multiple consistency criteria: It is well known that the strong consistency models have lower performance than the weak consistency models in terms of availability, latency, and scalability [3], [4], [9]. Most of cloud storage provide the eventual consistency forms [3]. However, even if the eventual consistency is sufficient for a large panel of applications some others require strong consistency criteria [10]. Moreover, a strong consistency is a desirable property for application programmers as it is easy to reason about. As a consequence, several systems target strong consistency while others aims at adding strong consistency features to provide multiple consistency criteria in their system [11], [12]. Mistore follows this vision by giving the possibility to choose the consistency criteria adapted to the application needs.

III. System Model

Mistore targets personal data (*e.g.*, documents, pictures, videos, music, etc). In the following, these data will be called *objects*. Communications are assumed to be reliable, and messages between two nodes are assumed to be received in the order they have been sent. POPs are interconnected via highly available and redundant channels provided by the ISP. Network partitions at this level occur with a negligible probability as the IP core network can recover and restore its state to avoid service disruption due to links/nodes failures [13], [14]. So we assume a system with no network partitions between the POPs. We assume different failures models for POPs, HGs, and the datacenter. The model of failure for POPs and datacenter is crash recovery, *i.e.*, we suppose that they both recover from a consistent state by using stable storage. Conversely, the model of failures exhibited by HGs is fail stop. We assume that crash are eventually detected by an eventual perfect detector failure [15]. We leave for future work more aggressive failures, Byzantine failures and objects corruption.

IV. Mistore Architecture

We consider a DSL infrastructure operated by a single ISP (See Fig. 1), whose main components are as follows.

The home gateways are connected in a client/server mode to the POP aggregating their xDSL lines and form its *region*. They communicate with the storage system via this POP. A fraction of the storage capacity of HGs is dedicated to the storage system. One part of this fraction stores some of the data that have been warehoused within the storage system. Note that these data do not necessarily belong to the owner of the HG. The other part of the fraction caches the data recently or frequently accessed by the HG owner. *The Points of Presence* provide several functionalities. They cache some data that transit through them, implement the data consistency and replication strategy of the storage system, and monitor the HGs they aggregate. Monitoring data (*e.g.*, storage capacity, data stored, availability state, etc.) are periodically transferred

to datacenters. *The datacenter* main role is to collect and store metadata and usage patterns from the HGs and POPs. It also offers a backup functionality for objects when their primary POP is unavailable (See Section V).

We leave for future work our vision to leverage the DCs knowledge on the system to implement multi- and auto-tiered storage features which can refer to an automated placement of data on nodes to optimize performance, availability, and recovery [16]. Indeed, we have nodes of different types and locations that could be distinguished in three tiers for storing cold, warm, and hot data while allowing data to move from one tier to another depending on data access patterns. *Cold data* are infrequently or never accessed, and thus could be efficiently and inexpensively handled by datacenters. *Warm data* are data that have been recently accessed in the storage system and present a higher probability to be accessed than cold data. They could be stored in regions from where their users access the storage system to reduce access latencies. Finally, *Hot data* are frequently accessed and could be stored in POPs and distributed like in a CDN. It would allow to answer users requests while avoiding to overload HGs that have low upload bandwidths or datacenters that exhibit large access latencies. Looking in this perspective, we recall that this paper focuses on the way Mistore deals data consistency and data replication on the HGs and the POPs.

V. Object consistency

Mistore replicates user objects to guarantee both their durability and availability. Now, as objects can be updated, consistency issues must be taken into account. Before diving into the different consistency criteria provided by Mistore, we briefly recall the main concepts and definitions that we will use. Traditionally consistency criteria are classified according to the fact that, given a distributed execution satisfying the considered criteria, it is always possible or not to build an history of this execution that could have been executed on a single processor and produce the same visible result. When this property is satisfied, the considered consistency criteria is said to be *strong*, otherwise it is *weak*. A strong consistency criteria requires stronger synchronization between processes that participates to the computation. This impacts the performance of the protocol but make the life of application developers easier. One can expect better performance of weak consistency criteria at the cost of more complex situations to deal with when manipulating data. Mistore implements both strong and weak consistency criteria. We rapidly expose the theory behind strong consistency criteria [17], and give a formal treatment of the weak consistency criteria implemented by Mistore.

A. Processes and operations

We consider a *concurrent system* composed of a set Π of n processes denoted $p_1, p_2, \ldots, p_n$ that cooperate through a finite set of *shared objects* X (*e.g.*, files in the case of Mistore). Each shared object $x \in X$ supports two types of operations: a `read` operation, and a `write` operation. The execution of writing value v into object x by process p_i is denoted by $w_i(x)v$. Conversely, the execution of reading value v from object x by process p_i is denoted $r_i(x)v$. We may omit the identity of the process that performs the operation when it is not relevant. To simplify we assume that each value written in an object is unique.[b]

B. History, legality and linear extension

Let $h_{i|_W}$ denotes the set of `write` operations performed by process p_i. Conversely let $h_{i|_R}$ denotes the set of `read` operations performed by process p_i. Let $h_i = h_{i|_R} \cup h_{i|_W}$ denotes the set of operations (both `read` and `write`) performed by process p_i. We assume that on a given process p_i only a single operation can occur at a given time. Hence operations in h_i are naturally totally ordered by an order relation that we denote $\rightarrow_i$. We call *local history* of p_i the set h_i ordered by $\rightarrow_i$. It is denoted by $\widehat{h_i} = (h_i, \rightarrow_i)$.

***Definition** 1 (Global history):* A global history $\widehat{H} = (H, \rightarrow_H)$ of a concurrent system (Π, X) is a set H partially ordered by $\rightarrow_H$, with $H = \bigcup_i h_i$ and $\rightarrow_H$ a partial order relation containing $\rightarrow_{rf}$ (that is $\rightarrow_{rf} \subseteq \rightarrow_H$), where $\rightarrow_{rf}$ is a partial order relation called *read from* order and defined as follows. For any two operations $op_1 \in H$ and $op_2 \in H$, we say that $op_1 \rightarrow_{rf} op_2$ if and only if

1) either $\exists i$ such that $op_1 \in h_i$ and $op_2 \in h_i$ and $op_1 \rightarrow_i op_2$,
2) or $\exists x, v$ such that $op_1 = w(x)v$ and $op_2 = r(x)v$,
3) or $\exists op_3 \in H$ such that $op_1 \rightarrow_{rf} op_3$ and $op_3 \rightarrow_{rf} op_2$.

***Definition** 2 (Linear extension of a global history):* Given a global history $\widehat{H} = (H, \rightarrow_H)$, a *linear extension* of $\widehat{H}$ and denoted $\vec{H} = (H, \rightarrow)$ is a total order $\rightarrow$ on H that is compatible with $\rightarrow_H$, that is for any two operations $op_1 \in H$ and $op_2 \in H$ if $op_1 \rightarrow_H op_2$ then $op_1 \rightarrow op_2$.

A linear extension of a global history $\widehat{H}$ can be seen as a topological sort of the directed acyclic graph[c] induced by the partial order relation $\rightarrow_H$.

We assume that an initial value has been written (by a fictitious `write` operation) in each variable. With this assumption, a linear history is *legal* if the following holds.

***Definition** 3 (Legality of a read):* Let $\vec{H} = (H, \rightarrow)$ be a linear extension. A `read` operation $op_r = r(x)v \in H$ is *legal* if and only if:

1) there exists a corresponding `write` operation $op_w = w(x)v \in H$ such that $op_w \rightarrow op_r$,

[b]This hypothesis can be easily implemented by assuming that each value written in an object is a triplet of the form of $(p_i, v, count_i)$ where $count_i$ is a counter maintained by process p_i counting its `write` operations.

[c]This graph $G = (V, E)$ is defined by its vertexes $V = H$ and there is a edge between two vertices $op_1 \in V$ and $op_2 \in V$ if and only if $op_1 \rightarrow_H op_2$.

2) and for all operations op such that $op_w \rightarrow op \rightarrow op_r$ $op \neq w(x)\star$.

***Definition** 4 (Legality of a linear extension):* A linear extension $\vec{H} = (H, \rightarrow)$ is *legal* if and only if all of its `read` operations are legal.

A linear history is legal when both its `read` and `write` operations respect the expected semantics of variables on a single threaded processor: a `read` operation on a variable returns the last value written in this variable.

C. Strong consistency criteria

***Definition** 5 (Sequential consistency):* A global history $H = (H, \rightarrow_H)$ is *sequentially consistent* if it admits a linear extension which is legal.

This consistency criterion is the simplest and weakest example of strong consistency criteria. It means that the concurrent execution could have happened on a single processor. This criteria has been first proposed by Lamport in [18].

We now present a stronger consistency criteria which brings into play physical real time. Each operation op happening on a process starts at a given real time denoted by $op.start$ and ends at a given real time denoted by $op.end$ (we have $op.start < op.end$). With these notations in hand, we can specify the partial order relation denoted by $\rightarrow_{rt}$ as follows.

***Definition** 6 (Real time precedence):* Let H be a set of operations, and let $op_1, op_2 \in H$ be any two operations. We say that op_1 precedes op_2 with respect to real time, which is denoted by $op_1 \rightarrow_{rt} op_2$, if and only if $op_1.end \leq op_2.start$.

***Definition** 7 (Real time concurrency):* We say that two operations op_1 and op_2 are *real-time concurrent*, which will be denoted by $op_1 \|_{rt} op_2$, if and only if neither $op_1 \rightarrow_{rt} op_2$, nor $op_2 \rightarrow_{rt} op_1$.

***Definition** 8 (Atomicity):* A global history $\widehat{H} = (H, \rightarrow_H)$ is atomically consistent if it admits a linear extension $\vec{H} = (H, \rightarrow)$ such that:

1) $\vec{H}$ is sequentially consistent,
2) and $\rightarrow_{rt} \subseteq \rightarrow$ (*i.e* $\rightarrow$ is compatible with $\rightarrow_{rt}$).

***Definition** 9 (Read/write order):* Let H be a set of operations, and let $op_1, op_2 \in H$ be any two operations. We say that op_1 precedes op_2 with respect to the **read/write order**, which will be denoted by $op_1 \rightarrow_{rw} op_2$, if and only if:

- either $op_1 \rightarrow_{rt} op_2$,
- or $op_1 \|_{rt} op_2$, $op_1 = r(x)\star$ and $op_2 = w(x)\star$, then
  1) $op_1 \rightarrow_{rw} op_2 \Leftrightarrow op_1.start \leq op_2.start$,
  2) $op_2 \rightarrow_{rw} op_1 \Leftrightarrow op_2.start < op_1.start$.

It is clear that $\rightarrow_{rt} \subseteq \rightarrow_{rw}$.

***Definition** 10 (Read/write mutual exclusion):* A global history $\widehat{H} = (H, \rightarrow_H)$ is compatible with the **read-write mutual exclusion** consistency criteria if it admits a linear extension $\vec{H} = (H, \rightarrow)$ such that $\rightarrow_{rw} \subseteq \rightarrow$.

D. Weak consistency criterion

From these criteria, we propose to define a weak consistency criteria, that we will call **monotonic read** consistency. To the best of our knowledge it is the first time that a weak criteria is defined in the same formalism framework as strong ones.

***Definition** 11 (Local history):* Let $\widehat{H} = (H, \rightarrow_H)$ be a global history. A local history $(H_i, \rightarrow_{H_i})$ with respect to process p_i is defined as follows :

- $H_i = h_i \cup \left(\bigcup_{j \neq i} h_{j|_W}\right)$,
- $\rightarrow_{H_i}$ is compatible with $\rightarrow_H$: for any operations op_1 and op_2 in H_i, we have $op_1 \rightarrow_i op_2 \Leftrightarrow op_1 \rightarrow op_2$.

***Definition** 12 (Read monotonic local history):* Let $\widehat{H} = (H, \rightarrow_H)$ be a global history. $\widehat{H}$ is consistent with the **read monotonic** consistency criteria if and only if for all processes $p_i \in \Pi$, the local history $(H_i, \rightarrow_{H_i})$ is **sequentially** consistent.

VI. Implementation

Mistore implements the different levels of consistency described in Section V through a lock service. Two types of locks exist. A *read lock* that ensures the mutual exclusion consistency criteria, and a *write lock* that ensures the SWMR model.

The lock service follows a primary-backup scheme [19]. A primary and backup lock server per object is implemented and are executed by the POPs. Fig. 2 and Fig. 3 show the pseudo-code executed by HGs and POPs to acquire a lock from respectively their associated POP and the primary POP of the concerned object. The lock on an object is created by function GRANTLOCK shown in Fig. 4. Function ISAVAILABLE checks if a requested lock on a specific object can be granted to a HG. To deal with nodes failures, a *lease* is associated with each granted lock [20], [21]. At *lease time*, that is, at the expiration time of the lock, a client has to renew the lease if the current read/write operation is not completed. This is achieved by contacting the primary in charge of the object. Note that for communication costs reasons, the lease time is only activated on the primary, not on the backup. Thus, the primary does not have to acknowledge the backup each time a lock lease is renewed. The backup activates the lease time of a lock only when it takes over this role upon detection of the primary failure.

A. Write operation

A write operation on an object creates a new version of this object in Mistore. When a client wants to write (update) an object o, it sends a request to the POP p associated to his HG as presented in Fig. 5 and Fig 6. If o is written for the first time, a primary and a backup POP are affected to it (See Fig 7). These POPs are responsible for the replication of that object and for its consistency management. The primary is the POP associated with the client HG issuing the creation request. The backup is randomly determined by the primary among the other POPs of the system (invocation of the GETBACKUP()

```
1: function ACQUIRELOCK(o_id, lockType)
2:     myPOP ← GETMYPOP()
3:     lock ← myPOP.ACQUIRELOCK(selfID, o_id, lockType)
4:     return lock
5: end function
```

Figure 2. Home gateway: Function ACQUIRELOCK.

```
1: function ACQUIRELOCK(hg_id, o_id, lockType)
2:     if self.ISPRIMARYOF(o_id) then
3:         lock ← GRANTLOCK(hg_id, o_id, lockType)
4:     else
5:         primary ← GETPRIMARYOF(o_id)
6:         lock ← primary.ACQUIRELOCK(hg_id, o_id, lockType)
7:     end if
8:     return lock
9: end function
```

Figure 3. Point of Presence: Function ACQUIRELOCK.

```
1: function GRANTLOCK(hg_id, o_id, lockType)
2:     lock ← null
3:     if typeOf(lockType) = readLock then
4:         if ¬ ISAVAILABLE(hg_id, o_id, writeLock) then
5:             lock ← INCREMENTLOCK(hg_id, o_id, readLock)
6:         end if
7:     else
8:         if ISAVAILABLE(hg_id, o_id, lockType) then
9:             lock ← LOCK(hg_id, o_id, lockType)
10:        end if
11:    end if
12:    return lock
13: end function
```

Figure 4. Point of Presence : Function GRANTLOCK.

function in Fig. 7). The create operation returns o_{id} the identifier of an object o, which is the concatenation of the identifiers of o primary, o backup, o HG, and a creation counter that represents the number of objects created by the HG that owns o. In the following the primary and the backup of object o are respectively denoted by o_p and o_b. When the primary of an object is unavailable, its backup becomes the primary and the datacenter becomes the new backup. When the primary recovers, it takes over its role. Our solution only makes use of one backup because of the high availability of these nodes [22]. A write operation on object on o that has already been created in the system must contain the write lock for o to respect the SWMR model. Note that as a client can crash or goes off before the end of a write operation, the write lock on o is maintained by POP p as long as the operation is not completed to ensure the SWMR model. POP p locally caches a copy of o and asks both the primary o_p and the backup o_b to replicate o. When o_p and o_b receive this request, they both locally cache o, acknowledge POP p (as described in Fig. 8), and trigger the replication operation on their HGs (see Lines 3–7 in Fig. 8, and Fig. 9). Upon receipt of both acknowledgements, POP p releases the write lock and notifies the client that the write operation is completed. Object o is then available for subsequent read and write operations, which makes the replication process on HGs transparent to clients.

```
1: function WRITE(obj, param)
2:     myPOP ← self.GETPOP()
3:     writeAck ← myPOP.WRITE(o, param)
4:     UPDATECACHE(writeAck.o_id, writeAck.o_version, o)
5:     return writeAck
6: end function
```

Figure 5. Home gateway: Function WRITE.

```
1: function WRITE(o, param)
2:     if ¬param.writeLock then
3:         o_id ← CREATE(o, param)
4:         o_newVersion ← 1
5:     else
6:         KEEPALIVE(param.writeLock)
7:         o_id ← param.writeLock.o_id
8:         o_newVersion ← param.writeLock.o_version + 1
9:     end if
10:    UPDATECACHE(o_id, o_newVersion, o)
11:    primary ← PRIMARYOF(o_id)
12:    primaryStoreAck ← primary.STORE(o, param)
13:    backup ← BACKUPOF(o_id)
14:    backupStoreAck ← backup.STORE(o, param)
15:    self.LOG_OUT(param, primaryStoreAck, backupStoreAck)
16:    return [o_id, o_newVersion]
17: end function
```

Figure 6. Point of Presence: Function WRITE.

```
1: function CREATE(o, param)
2:     backup ← GETBACKUP(o, param)
3:     o_id ← CREATEID(self, backup, param.HG, param.counter)
4:     return o_id
5: end function
```

Figure 7. Point of Presence: Function CREATE.

```
1: function STORE(o, param)
2:     UPDATECACHE(param.o_id, param.o_version, o)
3:     HGList ← GETHGLIST(param.availability)
4:     for each hg in HGList do
5:         storeAck ← hg.STORE(param.o_id, param.o_version, o)
6:         self.LOG_IN(param, storeAck)
7:     end for
8:     return [self, param.o_id, param.o_version]
9: end function
```

Figure 8. Point of Presence: Function STORE.

```
1: function STORE(o_id, o, o_version)
2:     self.WRITEONDISK(o_id, o, o_version)
3:     UPDATECACHE(o_id, o_version, o)
4:     return [self, o_id, o_version]
5: end function
```

Figure 9. Home Gateway: Function STORE.

B. Read operation

Mistore gives the opportunity to specify the consistency criteria that the read object must guarantee, or a specific version of desired object. Both choices are specified in the *version flag* parameter of the read operation. In addition, in the quest of reducing the latency of read operations, Mistore favors readings from local caches prior to propagating the request to both the primary and the backup of the object. Read operations handle two parameters, the identifier o_{id} of the requested object, and the *version flag* which indicates both the consistency criteria and the versioning view. The

```
1: function READ(o_id, versionFlag)
2:     if ISCACHED(o_id, versionFlag) then
3:         o ← self.READONDISK(o_id, versionFlag)
4:     else
5:         myPOP ← self.GETPOP()
6:         o ← myPOP.READ(o_id, versionFlag)
7:     end if
8:     UPDATECACHE(o_id, versionFlag.objVersion, obj)
9:     return o
10: end function
```

Figure 10. Home Gateway: Function READ.

```
1: function READ(o_id, versionFlag)
2:     if CHECKREADLOCK(versionFlag.readLock) then
3:         KEEPALIVE(versionFlag.readLock)
4:     end if
5:     if ISCACHED(o_id, versionFlag) then
6:         o ← self.READONDISK(o_id, versionFlag)
7:     else
8:         POPReplica ← RANDOM(primaryOf(o_id),
                                backupOf(o_id))
9:         o ← POPReplica.RETRIEVE(o_id, versionFlag)
10:    end if
11:    UPDATECACHE(o_id, versionFlag.o_version, o)
12:    return o
13: end function
```

Figure 11. Point of Presence: Function READ.

```
1: function RETRIEVE(o_id, versionFlag)
2:     o ← null
3:     if ISCACHED(o_id, versionFlag) then
4:         o ← self.READONDISK(o_id, versionFlag)
5:     else
6:         HGReplica ← GETHGREPLICA(o_id, versionFlag)
7:         o ← HGReplica.RETRIEVE(o_id, versionFlag)
8:     end if
9:     UPDATECACHE(o_id, versionFlag.o_version, o)
10:    return o
11: end function
```

Figure 12. Point of Presence: Function RETRIEVE.

```
1: function RETRIEVE(o_id, versionFlag)
2:     o ← self.READONDISK(o_id, versionFlag)
3:     UPDATECACHE(o_id, versionFlag.o_version, o)
4:     return o
5: end function
```

Figure 13. Home gateway: Function RETRIEVE.

different types of consistency criteria handled by Mistore are the read/write mutal exclusion criteria, the atomicity, and the eventual monotonic-read consistency one. If the read operation must be handled by the primary or the backup POPs then a single one is chosen (through a random choice). Figure 10 shows the code executed by HGs to handle a read operation, Figure 11 the code executed by a POP when it receives a read operation from a HG located in its region. Figure 12 shows the code executed by a POP to retrieve a requested object from its cache or from the HGs in its region and finally Figure 13 describes the code executed by a HG to retrieve an object from its local disk. In the following we present how the different consistency criteria are handled.

Read/write mutual exclusion: When a read operation is invoked with the read/write mutual exclusion flag, a read lock must be provided. When a POP receives this request from a HG, it maintains the read lock active until the requested object has been received by the HG that invoked the read operation. A read lock contains the last version number of an object o and ensures that as long as the read lock is active, no concurrent write operation on o will be executed. Thus, the read operation can read the object from the cache of any POP if it contains the last version of o, otherwise the request must be forwarded to the primary or the backup of the object.

Atomic consistency: When a read operation is invoked with the atomic consistency flag, no read lock must be provided. The read operation is forwarded to the primary or the backup of the object.

Eventual monotonic-read: When a read operation is invoked with the eventual monotonic-read flag, no read lock must be provided but the version flag must contain the most recent version number known by the client. The read operation can read the object from the cache of any POP if it contains a version equal to or newer than the version flag, otherwise the request must be forwarded to the primary or the backup of the object.

Versioning view: An object may be read from the cache of any POP or HG if it contains a version of the object equal to the one requested with the version flag, otherwise the request must be forwarded to the primary or the backup of the object.

TABLE I
NETWORK PARAMETERS USED IN SIMULATION

Network path	Upload (MB/s)	Download (MB/s)	Latency (ms)
HG - POP	5	20	75
POP1 - POP2	40000	40000	5
POP - DC	10000	10000	5

VII. EVALUATION

We have implemented Mistore on the Peersim simulator and performed all the simulations based on network parameters observed in an ISP infrastructure (see Table I).

Fig. 14 illustrates different write requests schemes. We observe that latencies are the lowest when objects are written synchronously on both their primary and backup POPs, the best case being when the write requests originate from the primary or the backup region. This gives us the insight that the primary and backup POPs of an object should be the POPs of the regions from which originate most of the write requests on the object. We also observe that writing synchronously an object in a remote HG shows the worst performance due to the low users network bandwidth. Both observations validate our choice to replicate objects on their primary and backup POPs in a synchronous way, and to replicate them asynchronously in the HGs of their regions. However the performance of writing objects in HGs can be considerably improved when a data stripping method is applied on objects (*i.e.* the object is fragmented in several blocks that are stored in parallel in

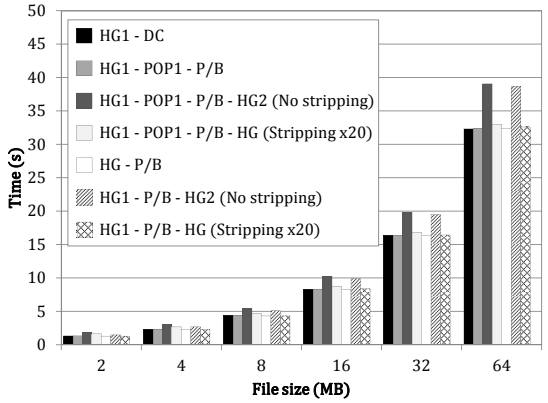

Figure 14. Write latencies. `HG-DC`: writing on the datacenter. `HG-POP-P/B`: writing an object on both its primary and backup POPs from a region different from these POPs region. `HG-POP-P/B-HG`: writing an object on HGs of both its primary and backup POPs from a region different from the primary and backup regions. `HG-P/B`: writing an object on both its primary and backup POPs from one these POPs region. `HG-P/B-HG)`: writing on HGs of both the primary and backup POPs of an object from one of these POPs regions. `No stripping`: an object replica is written entirely in one HG. `Stripping x20`: an object replica is fragmented in 20 blocks stored in parallel in 20 different HGs.

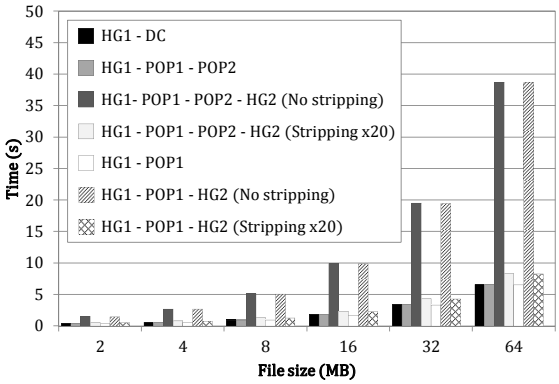

Figure 15. Read latencies. `HG-DC`: reading on the datacenter. `HG1-POP1-POP2`: reading on the POP of a region different from the one where the request is issued. `HG1-POP1-POP2-HG2`: reading on one HG connected to the POP of a region different from the one where the request is issued. `HG1-POP1`: reading on the POP of a region from where the request is issued. `No stripping`: the object is read entirely in 1 HG. `Stripping x20`: the 20 blocks stored in different HGs composing the object are read in parallel.

different HGs). Nevertheless we observe that gains obtained by using data stripping are more substantial on big size objects than on smaller ones.

Fig. 15 illustrates performances of different read requests scenarios on remote nodes. First, latencies are the lowest when objects are read on a POP, the best case being the one when this POP is the one of the region from where the read request comes. Reading an object entirely on a remote HG is in general slow due to the low upload bandwidth of users. However, the latencies can be reduced when objects are read from several HGs in parallel. This motivates to use data stripping methods when storing objects such that they can be retrieved by aggregating the bandwidth of several HGs. Nevertheless, as for write requests, data stripping is more efficient on big size data objects. These results give us some insights for cache replacement policies. Actually, differences, regarding latencies, between storing small objects on their primary and backup or synchronously in remote HGs is not very large so we believe that big objects should have the priority to be kept in cache. Moreover, small objects may also be stored synchronously in remote HGs without a big impact on performance. We have also evaluated the impact of HGs and POPs failures on write and read operation latencies. Prior to describing simulation results, we briefly present how the system reacts to failures.

Crash of HGs: When a HG fails by crashing, the list of objects it stores is retrieved from the index maintained by the POP in its region. To keep a given degree of redundancy, these objects may need to be recovered from other replicas and replicated in another HG. The results of the pending requests of a lost HG are cached in the POP in its region. Thus, when the HG recovers, acknowledgments are sent to the HG for write requests, and read requests are satisfied by accessing the cache of the POP in the region.

Crash of POPs: The failure of a POP means no Internet connection for the clients in its region. Thus we cannot not evaluate read and write latencies for users in a region where the POP is unavailable. Moreover, the crash of a POP means the crash of the primary or the backup of several objects. Without the primary of an object, its consistency can not be ensured so a new primary must be elected. As soon as the crash of a primary of an object is detected, its backup becomes the new primary and the datacenter becomes the new backup when an operation requiring a lock is activated. The backup can then trigger the lease of the locks it holds and takes over the service to the point where the old primary crashed. A crash of a POP is considered to be transient so a primary takes over the service when it recovers.

Fig. 16 shows that during a write operation, if either the primary or the backup POP of the object becomes available, the operation can finish without incurring a large latency overhead. This is due to the fact that the replication occurs in the IP core network, and thus the low user bandwidth is not involved when an object has to be replicated to a datacenter due to the unavailability of one of its primary or backup POP.

Fig. 17 shows the results of read operations during POPs and HGs unavailability. As argued before, we do not plot scenarios where the closest POP is unavailable — as it cuts users Internet connection — nor scenarios where the primary or the backup is unavailable — it does not incur any overhead as the object may be read on the available primary or backup POP. Similarly, we do not plot scenarios where HGs are unavailable since the POP knows which HGs are available and then will choose the available HGs to retrieve the requested objects. Now, when the POP (primary or backup) becomes unavailable after retrieving objects from the HGs and before forwarding them to their

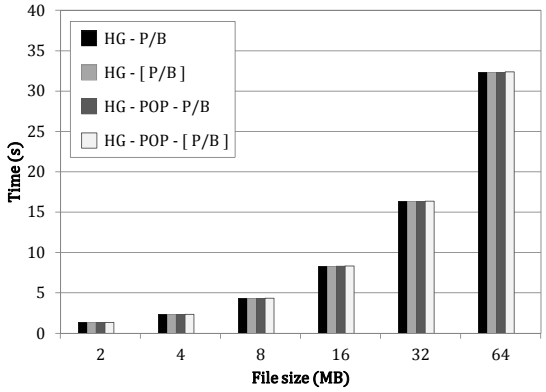

Figure 16. Write latencies in presence of crashes. See Fig. 14 for meanings of `HG-P/B` and `HG-POP-P/B`. `HG-[P/B]`: writing from the primary POP region of an object and the primary or the backup POP is unavailable. Notation "[.]" refers to the failed element. `HG-POP-[P/B]`: writing of an object from a region different from its primary and backup regions while either the primary or the backup POP is unavailable.

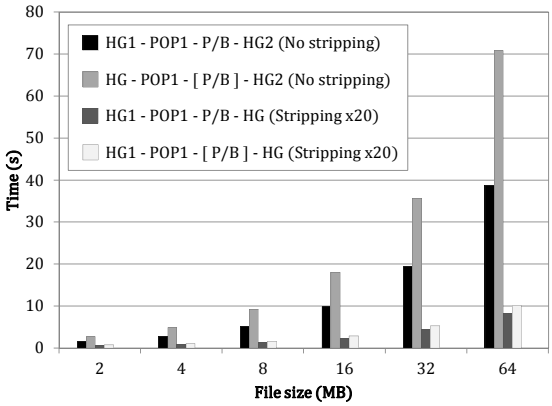

Figure 17. Read latencies in presence of crashes. `HG1-POP1-P/B-HG2 (No stripping)` and `HG1-POP1-P/B-HG2 (Stripping x20)`, see Fig. 15 for the meanings. `HG1-POP1-P/B-[HG2] (No Stripping)`: attempt to read an object in a HG located in a remote region but this remote HG is unavailable. `HG1-POP1-[P/B]-HG2 (No Stripping)`: attempt to read an object in a remote HG located in a remote region but the POP of that region (primary or backup) becomes unavailable before transmitting the retrieved object. `HG1-POP1-[P/B]-HG2 (Stripping x20)`: the same as the previous one but the object is retrieved by aggregating the object from 20 HGs in parallel before their associated POP becomes unavailable.

requester, we can observe that the overhead regarding latencies is very large when objects are not stripped over several HGs.

VIII. Conclusion and Future work

In this paper we have presented the main principles of Mistore, a distributed storage system to backup and share content dedicated to users who access Internet via a DSL technology. For future work, we plan to bring more storage intelligence close to the users (*i.e.* in POPs or/and HGs). We also seek to integrate efficient placement algorithms to minimize the amount of metadata to maintain in order to place/locate objects in Mistore. We are convinced that our architecture could also be suitable for user-generated content distribution and storage costs reduction via a multi-tiered data management. Evaluation of those points are other directions for future work.

References

[1] V. Valancius, N. Laoutaris, L. Massoulié, C. Diot, and P. Rodriguez, "Greening the internet with nano data centers," in *Procs of the International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[2] G. Pallis and A. Vakali, "Insight and perspectives for content delivery networks," *Communications of the ACM*, vol. 49, no. 1, 2006.

[3] W. Vogels, "Eventually consistent," *Commununications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[4] D. J. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer*, vol. 45, pp. 37–42, 2012.

[5] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2008.

[6] Wuala. http://www.wuala.com.

[7] Y. Sun, F. Liu, B. Li, B. Li, and X. Zhang, "FS2You: Peer-assisted semi-persistent online storage at a large scale," in *Procs of the International IEEE INFOCOM Conference*, 2009.

[8] C. Huang, A. Wang, J. Li, and K. W. Ross, "Understanding hybrid CDN-P2P: why limelight needs its own Red Swoosh," in *Procs of the International ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2008.

[9] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys*, vol. 37, no. 1, pp. 42–81, 2005.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Procs of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[11] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *Procs of theACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[12] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi, "Serializability, not serial: concurrency control and availability in multi-datacenter datastores," *Procs of the VLDB Endowment Journal*, vol. 5, no. 11, pp. 1459–1470, 2012.

[13] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second igp convergence in large ip networks," *ACM SIGCOMM Computers Communication Review*, vol. 35, no. 3, pp. 35–44, 2005.

[14] G. Iannaccone, C.-N. Chuah, S. Bhattacharyya, and C. Diot, "Feasibility of ip restoration in a tier 1 backbone," *IEEE Network*, vol. 18, no. 2, 2004.

[15] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[16] M. K. Aguilera, K. Keeton, A. Merchant, K.-K. Muniswamy-Reddy, and M. Uysal, "Improving recoverability in multi-tier storage systems," in *Procs of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[17] M. Raynal and A. Schiper, "A suite of formal definitions for consistency criteria in distributed shared memories," in *Procs of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, 1996.

[18] L. Lamport, "Concurrent reading and writing," *Communications of the ACM*, vol. 20, no. 11, pp. 806–811, 1977.

[19] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Procs of the 2nd international conference on Software engineering (ICSE)*, 1976.

[20] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system," in *Procs of the ACM symposium on Operating systems principles (SOSP)*, 1997.

[21] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in *Procs of the ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[22] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Procs of the ACM SIGCOMM Conference*, 2011.